

Demand Paging

- 6C Demand Paging
- 6D Replacement Algorithms
- 6E Thrashing and Working Sets
- 6F Optimizations and Advanced Technologies
- 6I Virtual Memory and I/O
- 6H Paging and Segmentation

Pure Swapping

- each segment is contiguous
 - in memory, and on secondary storage
 - all in memory, or all on swap device
- swapping takes a great deal of time
 - transferring entire data (and text) segments
- swapping wastes a great deal of memory
 - processes seldom need the entire segment
- variable length memory/disk allocation
 - complex, expensive, external fragmentation

Virtual Memory and Paging

2

Swapping is Wasteful

- process does not use all its pages all the time
 - code and data both exhibit *reference locality*
 - some code/data may seldom be used
- keeping all pages in memory wastes space
 - more space/process = fewer processes in memory
- swapping them all in and out wastes time
 - longer transfers, longer waits for disk
- it arbitrarily limits the size of a process
 - process must be smaller than available memory

Virtual Memory and Paging

3

Loading Pages “On Demand”

- paging MMU supports *not present* pages
 - CPU access of *present* pages proceeds normally
- accessing *not present* page generates a trap
 - operating system can process this “*page fault*”
 - recognize that it is a request for another page
 - read that page in and resume process execution
- entire process needn’t be in memory to run
 - start each process with a subset of its pages
 - load additional pages as program *demand*s them

Virtual Memory and Paging

4

Page Fault Handling

- initialize page table entries to *not present*
- CPU faults when invalid page is referenced
 1. trap forwarded to page fault handler
 2. determine which page, where it resides
 3. find and allocate a free page frame
 4. block process, schedule I/O to read page in
 5. update page table point at newly read-in page
 6. back up user-mode PC/regs to retry failed instruction
 7. unblock process, return to user-mode
- Meanwhile, other processes can run

Virtual Memory and Paging

5

Demand Paging – advantages

- improved system performance
 - fewer in-memory pages per process
 - more processes in primary memory
 - more parallelism, better throughput
 - better response time for processes already in memory
 - less time required to page processes in and out
 - less disk I/O means reduced queuing delays
- fewer limitations on process size
 - process can be larger than physical memory
 - process can have huge (sparse) virtual space

Virtual Memory and Paging

6

Are Page Faults a Problem?

- Page faults should not affect correctness
 - after fault is handled, desired page is in RAM
 - process runs again, and can now use that page (assuming the OS properly saves/restores state)
- But programs might run very slowly
 - additional context switches waste available CPU
 - additional disk I/O wastes available throughput
 - processes are delayed waiting for needed pages
- We must minimize the number of page faults

Minimizing Number of Page Faults

- There are two ways:
 - keep the “right” pages in memory
 - give a process more pages of memory
- How do we keep “right” pages in memory?
 - we have no control over what pages we bring in
 - but we can decide which pages to evict
 - this is called “replacement strategy”
- How many pages does a process need?
 - that depends on which process and when
 - this is called the process’ “working set”

Virtual Memory and Paging

8

Belady's Optimal Algorithm

- Q: which page should we replace?
A: the one we won't need for the longest time
- Why is this the right page?
 - it delays the next page fault as long as possible
 - minimum number of page faults per unit time
- How can we predict future references?
 - Belady cannot be implemented in a real system
 - but we can simulate it for test reference streams
 - we can compare other algorithms against it

Virtual Memory and Paging

9

Do we need an “Optimal” algorithm?

- Be very clear on what our goal is
 - we are not trying minimize the # of page faults
 - we are trying to minimize the cost of paging
- TANSTAAFL
 - we pay a price for every page fault
 - we also pay for every replacement decision
 - some decisions are more expensive than faults
 - we must do a cost/benefit analysis

Virtual Memory and Paging

10

Approximating Optimal Replacement

- note which pages have recently been used
 - use this data to predict future behavior
- Possible replacement algorithms
 - random, FIFO: straw-men ... forget them
- Least Recently Used
 - assert near future will be like recent past
 - programs do exhibit temporal and spatial locality
 - if we haven't used it recently, we probably won't soon
 - we don't have to be right 100% of the time
 - the more right we are, the more page faults we save

Virtual Memory and Paging

11

Why Programs Exhibit Locality

- Code locality
 - code in same routine is in same/adjacent page
 - loops iterate over the same code
 - a few routines are called repeatedly
 - intra-module calls are common
- Stack locality
 - activity focuses on this and adjacent call frames
- Data reference locality
 - this is common, but not assured

Virtual Memory and Paging

12

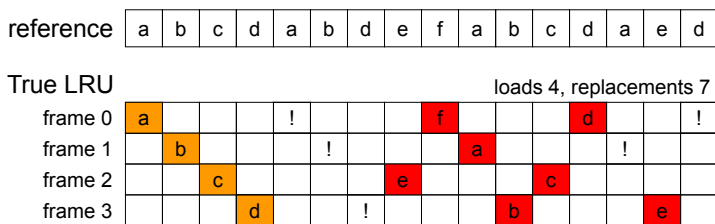
True LRU is hard to implement

- maintain this information in the MMU?
 - MMU notes the time, every time a page is referenced
 - maybe we can get a per-page read/written bit
- maintain this information in software?
 - mark all pages invalid, even if they are in memory
 - take a fault the first time each page is referenced
 - then mark this page valid for the rest of the time slice
- finding oldest page is prohibitively expensive
 - 16GB memory / 4K page = 4M pages to scan

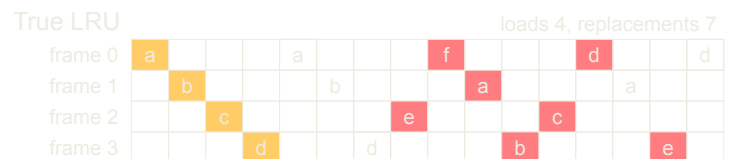
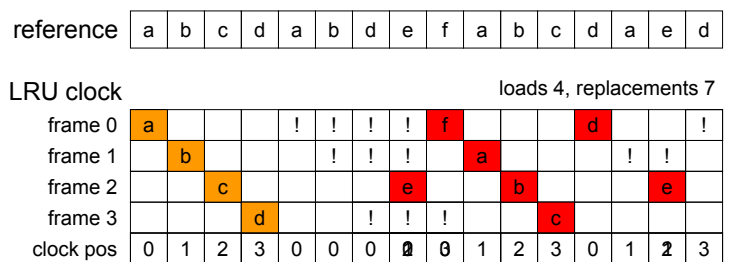
Practical LRU surrogates

- must be cheap
 - can't cause additional page faults
 - avoid scanning the whole page table (it is big)
- clock algorithms ... a surrogate for LRU
 - organize all pages in a circular list
 - position in the list is a surrogate for age
 - progressive scan whenever we need another page
 - for each page, ask MMU if page has been referenced
 - if so, reset the reference bit in the MMU; skip page
 - if not, consider this page to be the least recently used

True Global LRU Replacement



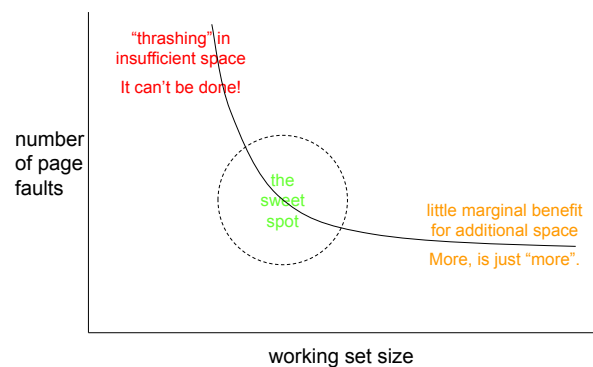
LRU Clock Algorithm



Working Sets – per process LRU

- Global LRU is probably a blunder
 - bad interaction with round-robin scheduling
 - better to give each process it's own page pool
 - do LRU replacement within that pool
- fixed # of pages per process is also bad
 - different processes exhibit different locality
 - which pages are needed changes over time
 - number of pages needed changes over time
 - much like different natural scheduling intervals
- we clearly want dynamic working sets

“natural” working set size



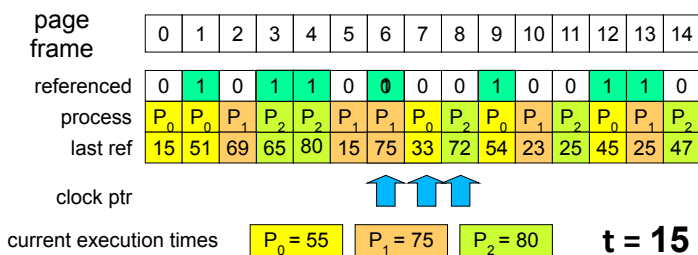
(Optimal Working Sets)

- What is optimal working set for a process?
 - number of pages needed during next time slice
- what if try to run process in fewer pages?
 - needed pages replace one another continuously
 - this is called "thrashing"
- how can we know what working set size is?
 - by observing the process behavior
- which pages should be in the working-set?
 - no need to guess, the process will fault for them

Implementing Working Sets

- managed working set size (the obvious approach)
 - assign page frames to each in-memory process
 - processes page against themselves in working set
 - observe paging behavior (faults per unit time)
 - adjust number of assigned page frames accordingly
- dynamic equilibrium (page stealing)
 - track last use time for each page, for owning process
 - find page least recently used (by its owner)
 - processes that need more pages tend to get more
 - processes that don't use their pages tend to lose them

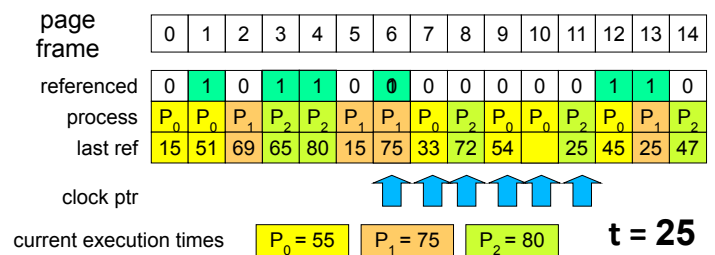
Working Set Clock Algorithm



P₀ gets a fault
 page 6 recently referenced
 clear ref bit, update time
 page 7 is (55-33=22) ms old
 P₀ replaces his own page



Working Set Clock Algorithm



P₀ gets a fault
 page 6 recently referenced
 page 7 is (55-33=22) ms old
 P₀ replaces his own page

P₀ gets a fault
 page 6 recently referenced
 page 7 is (55-33=22) ms old
 page 8 is (80-72=8) ms old
 page 9 is (55-54=1) ms old
 page 10 is (75-23=52) ms old
 P₀ steals this page from P₁



Thrashing Prevention

- working set size characterizes each process
 - how many pages it needs to run for τ milliseconds
- What if we don't have enough memory?
 - sum of working sets exceeds available memory
- we cannot squeeze working set sizes
 - this will result in thrashing
- reduce number of competing processes
 - swap some of the ready processes out
 - to ensure enough memory for the rest to run
- we can round-robin who is in and out

Clean and Dirty Pages

- consider a page, recently paged in from disk
 - there are two copies, one on disk, one in memory
- if the in-memory copy has not been modified
 - there is still a valid copy on disk
 - the in-memory copy is said to be "clean"
 - we can replace page w/o writing it back to disk
- if the in-memory copy has been modified
 - the copy on disk is no longer up-to-date
 - the in-memory copy is said to be "dirty"
 - if we write it out to disk, it becomes "clean" again

preemptive page laundering

- clean pages can be replaced at any time
 - copy on disk is already up to date
 - clean pages give flexibility to memory scheduler
 - many pages that can, if necessary, be replaced
- ongoing background write-out of dirty pages
 - find and write-out all dirty, non-running pages
 - no point in writing out a page that is actively in use
 - on assumption we will eventually have to page out
 - make them clean again, available for replacement
- this is the outgoing equivalent of pre-loading

Copy on Write

- *fork(2)* is a very expensive operation
 - we must copy all private data/stack pages
 - sadly most will be discarded by next *exec(2)*
- assume child will not update most pages
 - share all private pages, mark them *copy on write*
 - change them to be read-only for parent and child
 - on write-page fault, make a copy of that page
 - on *exec*, remaining pages become private again
- *copy on write* is a common optimization

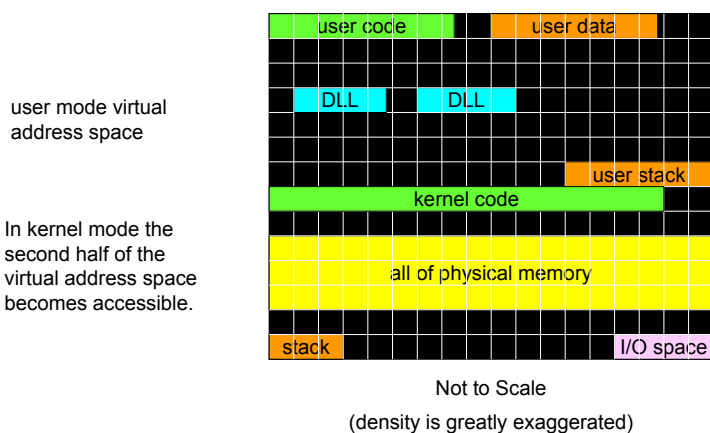
Virtual Memory and I/O

- user I/O requests use virtual buffer address
 - how can a device controller find that data
- kernel can copy data into physical buffers
 - accessing this data by its physical address
- kernel may translate virtual to physical
 - give device the corresponding physical address
 - pages must be locked in memory until complete
- CPU may include an I/O MMU
 - use page tables to translate virt addrs to phys
 - all DMA I/O references go through the I/O MMU

Kernel Space vs. User Space

- user space
 - multiple segments: code, data, stack, DLLs
 - segments are allocated in one-page units
 - data space managed as heap by user-mode code
- Kernel space (may be virtual or physical)
 - also includes all system code and data structures
 - also includes mapped I/O space
- physical memory divided into two classes
 - most managed as pages, for use by processes
 - some managed as storage heap for kernel allocation

Typical Kernel Virtual Address Space



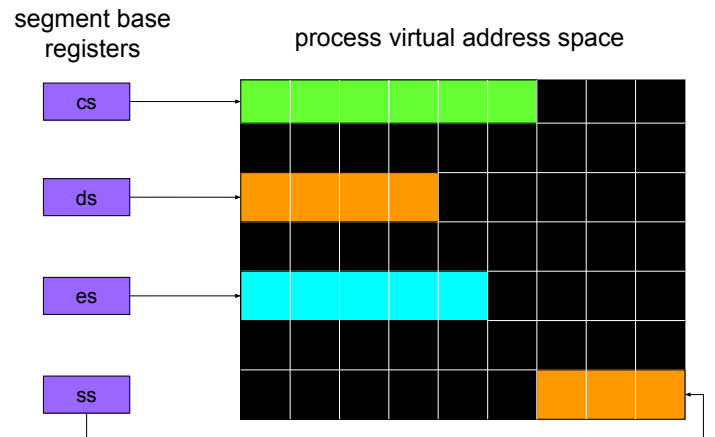
Moving Data between Kernel/User

- kernel often needs to access user data
 - to access system call parameters
 - to perform read and write system calls
- kernel may run in a virtual address space
 - which includes current process' address space
- special instructions for cross-space access
 - e.g. "move from previous data"
- kernel may execute w/physical addresses
 - software translation of user-space addresses

paging and segmentation

- pages are a very nice memory allocation unit
 - they eliminate internal and external fragmentation
 - they admit of a very simple and powerful MMU
- they are not a particularly natural unit of data
 - programs are comprised of, and operate on, segments
 - segments are the natural “chunks” of virtual address space
 - e.g. we map a new segment into the virtual address space
 - each code, data, stack segment contains many pages
- two levels of memory management abstraction
 - a virtual address space is comprised of segments
 - relocation & swapping is done on a page basis
 - segment base addressing, with page based relocation
- user processes see segments, paging is invisible

segmentation on top of paging



Segments – collections of pages

- a segment is a named collection of pages
 - each page has a home on secondary storage
- operations on segments:
 - create/open/destroy
 - map/unmap segment to/from process
 - find physical page number of virtual page n
- connection between paging & segmentation
 - segment mapping implemented w/page mapping
 - page fault uses segments to find requested page

Managing Secondary Storage

- where do pages live when not in memory?
 - we swap them out to secondary storage (disk)
 - how do we manage our swap space?
- as a pool of variable length partitions?
 - allocate a contiguous region for each process
- as a random collection of pages?
 - just use a bit-map to keep track of which are free
- as a file system?
 - create a file per process (or segment)
 - file offsets correspond to virtual address offsets

Paging and Shared Segments

- shared memory, executables and DLLs
- created/managed as mappable segments
 - one copy mapped into multiple processes
 - demand paging same as with any other pages
 - 2ndary home may be in a file system
- shared pages may not fit working set model
 - may not be associated with just one process
 - global LRU may be more appropriate
 - shared pages often need/get special handling

Reading and Assignments

Reading:

- Arpaci C27-27.2 ... thread APIs (s.b. review)
- Kampe: user-mode threads
- Kampe: introduction to Inter-Process-Communication
- named pipes, send(2), recv(2), mmap(2) system calls

Projects:

- we will help w/project 4A problems in the lab